WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Terms	Documents
l6 and tile	1

Database: US Patents Full-Text Database
US Pre-Grant Publication Full-Text Database
JPO Abstracts Database
EPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L7

Refine Search

Recall Text | Clear

Search History

DATE: Wednesday, February 19, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L7	l6 and tile	1	L7
L6	L5 and l1	1	L6
L5	L2 same texture same (color or colour) same (depth or z-buffer)	2	L5
L4	L1 and l3	25	L4
L3	L2 and texture and (color or colour) and (depth or z-buffer)	40	L3
L2	(uma or unif$4 near2 (memory or cache))	1723	L2
L1	graphic$3 adj2 (core or module or section or unit or block or circuit$3 or engine)	4318	L1

END OF SEARCH HISTORY

WEST

Generate Collection Print

L4: Entry 10 of 25 File: USPT Nov 13, 2001

DOCUMENT-IDENTIFIER: US 6317134 B1
TITLE: System software for use in a graphics computer system having a shared system memory and supporting DM Pbuffers and other constructs aliased as DM buffers

Abstract Text (1):
A computer system having a shared system memory, and system software in the computer system, are described herein. The computer system has a general purpose, shared system memory that is used for all processing, including video input/output operations, image conversion operations, and rendering operations. In operation, a digital media (DM) Pbuffer is created. The DM Pbuffer is aliased as a DM buffer in the system memory. This is done by storing in a color buffer identifier of the DM Pbuffer an identifier of the DM buffer. Thereafter, all graphical rendering operations directed to the DM Pbuffer are actually performed using the DM buffer.

Parent Case Text (4):
"Unified Memory Computer Architecture With Dynamic Graphics Memory Allocation," Ser. No. 08/713,779, filed Sep. 13, 1996 (now U.S. Pat. No. 6,104,417), incorporated herein by reference in its entirety.

Brief Summary Text (20):
A CPU rendering I/O and memory module (CRIMM) performs graphical rendering operations. According to the invention, a digital media (DM) Pbuffer is created. The DM Pbuffer is aliased as a DM buffer in the system memory. This is done by storing in a color buffer identifier of the DM Pbuffer an identifier of the DM buffer. Thereafter, all graphical rendering operations directed to the DM Pbuffer are actually performed using the DM buffer.

Drawing Description Text (23):
FIG. 28 illustrates a texture mapping scenario involving a texture map object according to an embodiment of the invention; and

Detailed Description Text (7):
The computer system 302 also includes a central processing unit (CPU) 304, and a video imaging and compression engine or module (VICM) 326. The VICM 326 performs image conversion functions, which are functions that convert at least a part of an image from a first form to a second form. Image conversion functions include video imaging and compression functions, such as data compression, data decompression, color space conversion, etc. The VICM 326 is connected to system memory 306, and has direct access to system memory 306. This differs from conventional computer systems, where the data compression/decompression modules are not connected to system memory (see FIG. 1, for example).

Detailed Description Text (8):
The computer system 302 also includes a CPU rendering I/O and memory engine or module (CRIMM) 308. The CRIMM 308 controls the system memory 306. The CRIMM 308 also performs graphical operations, such as rendering operations, and is also herein called a graphics rendering unit. The CRIMM 308 is connected to the system memory 306, and has direct access to system memory 306.

Detailed Description Text (10):
The CRIMM 308 is not directly connected to the GBEM 310. This is unlike conventional computer systems, where the graphics rendering engine is connected to the graphics back end via a frame buffer, which is dedicated to graphics operations (see FIG. 1). In the present invention, the CRIMM 308 is connected to the GBEM 310 via system memory 306, which is general purpose memory.

Detailed Description Text (103):
FIG. 20 illustrates a block diagram of the CRIMM 308. The CRIMM 308 includes a graphics rendering module 2002, which performs graphics rendering operations. Preferably, the graphics rendering module 2002 is compatible with OpenGL, although the present invention is not limited to this embodiment Instead, the present invention is adapted and intended to work with non-OpenGL systems, such as (but not limited to) QUICKDRAW 3D, DIRECT 3D, X WINDOWS, etc.

Detailed Description Text (107):
A graphics context defines the graphical configuration, characteristics, and/or capabilities of a drawable (described below) to which rendering operations are to be directed. A graphics context includes a set of graphical state variables. These state variables define the graphical configuration, characteristics, and/or capabilities of a drawable. For example, these state variables define whether RGB (red, green, blue) rendering is to be performed, whether single or double buffering is being used, whether there is depth or no depth, whether there is alpha or no alpha, etc. An application may be associated with a number of graphics contexts. At any time, however, there is only one current or active graphics context.

Detailed Description Text (108):
In an embodiment of the invention, a new graphics context is created by first creating an instance of FB (frame buffer) config, which is a data structure, and which is an abstract representation of a graphics context FB config has parameters that are used to specify values for graphical characteristics and capabilities. For example, FB config has parameters that define whether RGB rendering is to be performed, whether single or double buffering is being used, whether there is depth or no depth, whether there is alpha or no alpha, etc.

Detailed Description Text (112):
A Pbuffer is a drawable. Thus, a Pbuffer is an object to which rendering operations are directed. Referring to FIG. 26, a Pbuffer 2602 includes a color buffer 2604, an accumulation buffer 2606, and a Z buffer 2608. The color image being rendered is stored in the color buffer 2604. Depth information is stored in the Z buffer 2608. The accumulation buffer 2606 is used to accumulate images that are drawn into the color buffer 2604.

Detailed Description Text (116):
The present invention solves these problems by introducing a new type of Pbuffer, called a DM (digital media) Pbuffer 2702 (see FIG. 27). Like the Pbuffer 2602 shown in FIG. 26, the DM Pbuffer 2702 has an accumulation buffer 2606 and a Z buffer 2608. The DM Pbuffer 2702 of the present invention, however, has a color buffer identifier 2704, instead of a color buffer 2604. The color buffer identifier 2704 is a pointer to a DM buffer. More particularly, the color buffer identifier 2704 is preferably a storage location having stored therein an identifier (such as a pointer) to a DM buffer. The image that is being rendered is stored in this DM buffer, as opposed to being stored in the DM Pbuffer 2702 itself The DM Pbuffer is said to be associated with the DM buffer identified by the color buffer identifier 2704.

Detailed Description Text (117):
An DM Pbuffer can be (and typically is) associated with different DM buffers over time. This is done by changing the value of the color buffer identifier 2704 in the DM Pbuffer to address different DM buffers.

Detailed Description Text (118):
In an alternate embodiment, the color buffer identifier 2704 points to a plurality of DM buffers.

Detailed Description Text (120):
In operation, graphical operations are directed to a DM Pbuffer 2702. However, the CRIMM 308 does not attempt to perform those graphical operations on color image data stored in the DM Pbuffer 2702 itself. Instead, the CRIMM 308 performs the graphical operations on data in a DM buffer that is linked to (i.e., associated with) the DM Pbuffer 2702. As described above, a DM Pbuffer is associated with a DM buffer via the color buffer identifier 2704 in the DM Pbuffer. The DM Pbuffer 2702 is said to be aliased as the DM buffer which is identified by the color buffer identifier 2704.

Detailed Description Text (122):
A newly created DM Pbuffer 2702 has a color buffer identifier 2704 (FIG. 27), but this color buffer identifier 2704 does not initially store an address to a DM buffer. It is

necessary to called a function called glxDMBufferAssociate (that is part of the software libraries 1810), also called glXAssociateDMPbuffer in some implementations, to do this. This function receives as passed parameters an identifier of a DM Pbuffer 2702, and an identifier of a DM buffer. The function stores the identifier of the DM buffer into the color buffer identifier field 2704 of the DM Pbuffer 2702 (that is identified by the passed parameter).

Detailed Description Text (127):
The operation of the present invention shall now be further described by reference to FIG. 22, which illustrates a scenario 2202 where the CRIMM 308 is rendering to a drawable 2114A. The drawable 2114A is a DM Pbuffer, and its color buffer identifier 2704 identifies a DM buffer 2104A in the user application memory 2102. A flowchart 2302 in FIG. 23 depicts the operational steps associated with the scenario 2202 of FIG. 22.

Detailed Description Text (149):
Texture Mapping

Detailed Description Text (150):
DM buffers can be used in a great many other contexts and applications. Texture mapping is one such application.

Detailed Description Text (151):
In an embodiment of the invention, a texture map object 2804 representative of a texture includes a texture identifier 2806 (see FIG. 28). The texture map object 2804 does not itself store texture data. Instead, the texture identifier 2806 points to a DM buffer 2808 that stores texture data More particularly, the texture identifier 2806 is a memory location that stores a pointer to the DM buffer 2808 having texture data stored therein. The texture map object 2804 is said to be associated with, or aliased as, the DM buffer 2808. During texture mapping operations, the texture data in the DM buffer 2808 is used. This DM buffer 2808 is accessed by reference to the texture identifier 2806 in the texture map object 2804.

Detailed Description Text (152):
An example operation of this embodiment of the invention is represented by a flowchart 2902 in FIG. 29. In step 2906, data is generated and stored in a DM buffer 2808 that is associated with a DM Pbuffer 2816 (this association is graphically indicated by dotted line 2820). Such data in the DM buffer 2808 may be the result of multiple graphics rendering and/or image conversion operations which are initiated by a user application. As should be clear from the following discussion, the data in the DM buffer 2808 need not be copied from one place to another in order to use it for texture mapping.

Detailed Description Text (153):
In step 2908, the user application creates a Texture Map Object 2804.

Detailed Description Text (154):
In step 2910, the user application causes the pointer in the color buffer identifier 2818 of the DM Pbuffer 2816 to be copied to the texture identifier 2806 of the texture map object 2804. As a result, the texture map object 2804 becomes associated with the DM buffer 2808. In other words, the texture map object 2804 becomes associated with the texture data in the DM buffer 2808.

Detailed Description Text (155):
In step 2912, the user application causes the texture associated with the texture map object 2804 to be mapped to the image data associated with the DM Pbuffer 2810. According to the invention, the DM buffer 2808 is accessed by referenced to the texture identifier 2806 of the texture map object 2804. The texture data in the DM buffer 2808 is then mapped (in accordance with command(s) issued by the user application) to the ° image stored in the DM buffer 2814, which is accessed by reference to the color buffer identifier 2812 of the DM Pbuffer 2810.

CLAIMS:

2. The method of claim 1, wherein said DM Pbuffer comprises a color buffer identifier, and wherein step (2) comprises the step of:

storing in said color buffer identifier of said DM Pbuffer an identifier of said DM buffer.

5. The method of claim 1, further comprising the steps of:

(4) aliasing a texture map object as said DM buffer; and

(5) processing texture mapping operations involving said texture map object by using texture data stored in said DM buffer.

6. A computer system, comprising:

a system memory;

a central processing unit coupled directly to said system memory;

a graphics rendering unit coupled directly to said system memory and configured to process video images in said system memory; and

a user application executing in said central processing unit, said user application configured to cause,

a digital media (DM) Pbuffer to be created in said system memory

and configured to cause said DM Pbuffer to be aliased as a DM buffer in said system memory, said user application configured to invoke

graphical functions directed to said DM Pbuffer.

7. The computer system of claim 6, wherein said graphics rendering unit

is configured to render to said DM buffer in accordance with said invoked graphical function.

8. The computer system of claim 6, wherein said

user application is configured to use an identifier of said DM buffer to be stored in said color buffer identifier of said DM Pbuffer.

9. The computer system of claim 6,

wherein said user application is configured to alias a texture map object as said DM buffer

and to process texture mapping operations involving said texture map object by using texture data stored in said DM buffer.

10. A method of texture mapping, comprising the steps of:

(1) aliasing a texture map object as a first DM buffer, said first DM buffer storing texture data;

(2) aliasing a drawable as a second DM buffer, said second DM buffer storing image data; and

(3) mapping said texture map object to said drawable by mapping said texture data in said first DM buffer to said image data in said second DM buffer.

11. A system for texture mapping, comprising:

means for aliasing a texture map object as a first DM buffer, said first DM buffer storing texture data;

means for aliasing a drawable as a second DM buffer, said second DM buffer storing image data; and

means for mapping said texture map object to said drawable by mapping said texture data in said first DM buffer to said image data in said second DM buffer.

13. A computer program product comprising a computer useable medium having computer program logic recorded therein, said computer program logic enabling a processor in a computer to perform texture mapping in a system memory coupled to the processor, said computer program logic comprising:

a procedure that enables the processor to alias a texture map object as a first DM buffer in the system memory, said first DM buffer storing texture data;

a procedure that enables the processor to alias a drawable as a second DM buffer in the system memory, said second DM buffer storing image data; and

a procedure that enables the processor to map said texture map object to said drawable by mapping said texture data in said first DM buffer to said image data in said second DM buffer.

14. A computer system, comprising:

a system memory;

a central processing unit coupled directly to said system memory;

a graphics rendering unit coupled directly to said system memory and configured to process video images in said system memory; and

a user application executing in said central processing unit, said user application including,

means for causing a digital media (DM) Pbuffer to be created in said system memory,

aliasing means for causing said DM Pbuffer to be alised as a DM buffer in said system memory, and

means for invoking graphical functions directed to said DM Pbuffer.

WEST

Generate Collection Print

L4: Entry 7 of 25 File: USPT Jan 29, 2002

DOCUMENT-IDENTIFIER: US 6342892 B1
TITLE: Video game system and coprocessor for video game system

Abstract Text (1):
A low cost high performance three dimensional (3D) graphics system can model a world in three dimensions and project the model onto a two dimensional viewing plane selected based on a changeable viewpoint. The viewpoint can be changed on an interactive, real time basis by operating user input controls such as game controllers. The system rapidly produces a corresponding changing image (which can include animated cartoon characters or other animation for example) on the screen of a color television set. The richly featured high performance low cost system gives consumers the chance to interact in real time inside magnificent virtual 3D worlds to provide a high degree of image realism, excitement and flexibility. An optimum feature set/architecture (including a custom designed graphics/audio coprocessor) provides high quality fast moving 3D images and digital stereo sound for video game play and other graphics applications.

Brief Summary Text (6):
3D graphics are fundamentally different from 2D graphics. In 3D graphics techniques, a "world" is represented in three dimensional space. The system can allow the user to select a viewpoint within the world. The system creates an image by "projecting" the world based on the selected viewpoint. The result is a true three-dimensional image having depth and realism.

Brief Summary Text (11):
Optimum feature set/architecture for a low cost system for use with a color television set to provide video game play and other graphics applications in a low cost system and/or to produce particular screen effects

Brief Summary Text (13):
Signal processor sharing between graphics digital processing and audio signal processing to achieve high quality stereo sound and 3-D graphics in a low cost color television based system

Drawing Description Text (36):
FIG. 22 shows an example texture memory tile descriptor arrangement;

Drawing Description Text (37):
FIG. 23 shows an example texture unit process;

Drawing Description Text (38):
FIG. 24 shows an example texture coordinate unit and texture memory unit architecture;

Drawing Description Text (39):
FIG. 25 shows an example texture memory color index mode lookup;

Drawing Description Text (40):
FIG. 26 shows an example more detailed use of the texture memory to store color indexed textures;

Drawing Description Text (41):
FIG. 27 shows an example color combiner operation;

Drawing Description Text (46):
FIG. 32 shows an example color pixel format;

Drawing Description Text (47):

FIG. 33 shows an example depth (z) pixel format;

Detailed Description Text (2):
FIG. 1 shows an example embodiment video game system 50 in accordance with the present invention(s). Video game system 50 in this example includes a main unit 52, a video game storage device 54, and handheld controllers 56 (or other user input devices). In this example, main unit 52 connects to a conventional home color television set 58. Television set 58 displays 3D video game images on its television screen 60 and reproduces stereo sound through its loud speakers 62.

Detailed Description Text (4):
"Read only memory" chip 76 stores software instructions and other information pertaining to a particular video game. The read only memory chip 76 in one storage device 54 may, for example, contain instructions and other information for an adventure game. The read only memory chip 76 in another storage device 54 may contain instructions and information to play a driving or car race game. The read only memory chip 76 of still another storage device 54 may contain instructions and information for playing an educational game. To play one game as opposed to another, the user of video game system 50 simply plugs the appropriate storage device 54 into main unit slot 64--thereby connecting the storage device's read only memory chip 76 (and any other circuitry the storage device may contain) to the main unit 52. This enables the main unit 52 to access the information contained within read only memory 76, which information controls the main unit to play the appropriate video game by displaying images and reproducing sound on color television set 58 as specified under control of the video game software in the read only memory.

Detailed Description Text (5):
To play a video game using video game system 50, the user first connects main unit 52 to his or her color television set 58 by hooking a cable 78 between the two. Main unit 52 produces both "video" signals and "audio" signals for controlling color television set 58. The "video" signals are what controls the images displayed on the television screen 60, and the "audio" signals are played back as sound through television loudspeakers 62. Depending on the type of color television set 58, it may be necessary to use an additional unit called an "RF modulator" in line between main unit 52 and color television set 58. An "RF modulator" (not shown) converts the video and audio outputs of main unit 52 into a broadcast type television signal (e.g., on television channel 2 or 3) that can be received and processed using the television set's internal "tuner."

Detailed Description Text (11):
FIGS. 1A-1F show just one example of some three-dimensional screen effects that system 50 can generate on the screen of color television set 58. FIGS. 1A-1F are in black and white because patents cannot print in color, but system 50 can display these different screens in brilliant color on the color television set. Moreover, system 50 can create these images very rapidly (e.g., seconds or tenths of seconds) in real time response to operation of game controllers 86.

Detailed Description Text (12):
Each of FIGS. 1A-1F was generated using a three-dimensional model of a "world" that represents a castle on a hilltop. This model is made up of geometric shapes (i.e., polygons) and "textures" (digitally stored pictures) that are "mapped" onto the surfaces defined by the geometric shapes. System 50 sizes, rotates and moves these geometric shapes appropriately, "projects" them, and puts them all together to provide a realistic image of the three-dimensional world from any arbitrary viewpoint. System 50 can do this interactively in real time response to a person's operation of game controllers 86.

Detailed Description Text (14):
FIGS. 1D and 1E show views from the ground looking up at or near the castle main gate. System 50 can generate these views interactively in real time response to game controller inputs commanding the viewpoint to "land" in front of the castle, and commanding the "virtual viewer" (i.e., the imaginary person moving through the 3-D world through whose eyes the scenes are displayed) to face in different directions. FIG. 1D shows an example of "texture mapping" in which a texture (picture) of a brick wall is mapped onto the castle walls to create a very realistic image.

Detailed Description Text (18):
Main processor 100 generates, from time to time, lists of commands that tell the coprocessor 200 what to do. Coprocessor 200 in this example comprises a special purpose

high performance application-specific integrated circuit (ASIC) having an internal design that is optimized for rapidly processing 3-D graphics and digital audio. In response to commands provided by main processor 100 over path 102, coprocessor 200 generates video and audio for application to color television set 58. The coprocessor 200 uses graphics, audio and other data stored within main memory 300 and/or video game storage device 54 to generate images and sound.

Detailed Description Text (28):
Texture coordinate generation

Detailed Description Text (29):
Texture application and filtering

Detailed Description Text (30):
Color combining

Detailed Description Text (35):
FIG. 3 shows the main processes performed by the main processor 100, coprocessor 200 and main memory 300 in this example system 50. The main processor 100 receives inputs from the game controllers 56 and executes the video game program provided by storage device 54 to provide game processing (block 120). It provides animation, and assembles graphics and sound commands for use by coprocessor 200. The graphics and sound commands generated by main processor 100 are processed by blocks 122, 124 and 126--each of which is performed by coprocessor 200. In this example, the coprocessor signal processor 400 performs 3D geometry transformation and lighting processing (block 122) to generate graphics display commands for display processor 500. Display processor 500 "draws" graphics primitives (e.g., lines, triangles and rectangles) to create an image for display on color TV 58. Display processor 500 performs this "drawing" or rendering function by "rasterizing" each primitive and applying a texture to it if desired (block 126). It does this very rapidly--e.g., on the order of many millions of "pixels" (color television picture elements) a second. Display processor 500 writes its image output into a frame buffer in main memory 300 (block 128). This frame buffer stores a digital representation of the image to be displayed on the television screen 60. Additional circuitry within coprocessor 200 reads the information from the frame buffer and outputs it to television 58 for display (block 130).

Detailed Description Text (43):
The list of graphics commands is called a "display list" because it controls the images coprocessor 200 displays on the TV screen 60. The list of audio commands is called a "play list" because it controls the sounds that are played over loudspeaker 62. Generally, main processor 100 specifies both a new display list and a new play list for each video "frame" time of color television set 58.

Detailed Description Text (46):
The signal processor 400 can provide the graphics display commands 112 directly to display processor 500 over a path internal to coprocessor 200, or it may write those graphics display commands into main memory 300 for retrieval by the display processor (not shown). These graphics display commands 112 command display processor 500 to draw ("render") specified geometric shapes with specified characteristics (FIG. 4a, block 126a). For example, display processor 500 can draw lines, triangles or rectangles (polygons) based on these graphics display commands 112, and may fill triangles and rectangles with particular colors and/or textures 116 (e.g., images of leaves of a tree or bricks of a brick wall)--all as specified by the graphics display commands 112. Main processor 100 stores the texture images 116 into main memory 300 for access by display processor 500. It is also possible for main processor 100 to write graphics display commands 112 directly into main memory 300 for retrieval by display processor 500 to directly command the display processor.

Detailed Description Text (47):
Display processor 500 generates, as its output, a digitized representation of the image that is to appear on television screen 60 (FIG. 4A, block 126b). This digitized image, sometimes called a "bit map," is stored within a frame buffer 118 residing in main memory 300. Display processor 500 can also store and use a depth (Z) buffer 118b in main memory 300 to store depth information for the image. Another part of coprocessor 200 called the "video interface" (not shown) reads the frame buffer 118 and converts its contents into video signals for application to color television set 58 (FIG. 4a, block 127). Typically, frame buffer 118 is "double buffered," meaning that coprocessor 200 can be writing the "next" image into half of the frame buffer while the video interface is reading out the other half.

Detailed Description Text (57):
FIG. 5 also shows that the audio and video outputs of coprocessor 200 are processed by some electronics outside of the coprocessor before being sent to television set 58. In particular, in this example coprocessor 200 outputs its audio and video information in digital form, but conventional home color television sets 58 generally require analog audio and video signals. Therefore, the digital outputs of coprocessor 200 are converted into analog form--a function performed for the audio information by DAC 140 and for the video information by VDAC 144. The analog audio output of DAC 140 is amplified by an audio amplifier 142 that may also mix audio signals generated externally of main unit 52 and supplied through connector 154. The analog video output of VDAC 144 is provided to video encoder 146, which may, for example, convert "RGB" input signals to composite video outputs. The amplified stereo audio output of amplifier 142 and the composite video output of video encoder 146 are provided to home color television set 58 through a connector not shown.

Detailed Description Text (59):
FIG. 5 shows that storage device 54 also stores a database of graphics and sound data 112a needed to provide the graphics and sound of the particular video game. Main processor 100 reads the graphics and sound data 112a from storage device 54 on an as-needed basis and stores it into main memory 300 in the form of texture data 116, sound data 112b and graphics data 112c. In this example, display processor 500 includes an internal texture memory 502 into which the texture data 116 is copied on an as-needed basis for use by the display processor.

Detailed Description Text (70):
Unified Main Memory 300

Detailed Description Text (77):
Texture maps 116 and other graphics data 112c

Detailed Description Text (78):
Color image frame buffer 118a

Detailed Description Text (79):
Depth (z) buffer 118b

Detailed Description Text (85):
Advantages and disadvantages in using single address space memory architectures for raster scan display systems are known (see, for example, Foley et al, Computer Graphics: Principles and Practice at 177-178 (2d Ed. Addison-Wesley 1990). Many video game (and other graphics) system architects in the past rejected a single address space architecture in favor of using dedicated video RAM devices for graphics data and using other types of memory devices for other types of data. However, a unified main memory 300 provides a number of advantages in this particular example of a video game system 50. For example:

Detailed Description Text (89):
The Unified Memory Provides Memory Allocation Flexibility

Detailed Description Text (91):
For example, one video game programmer might provide a large frame buffer for high resolution images and/or image scrolling and panning, while another programmer may decide to use a smaller frame buffer so as to free up memory space for other data structures (e.g., textures or audio data). One application may devote more of main memory 300 storage for audio data structures and less to graphics data, while another application may allocate most of the storage for graphics related data. The same video game program 108 can dynamically shift memory allocation from one part of game play to another (e.g., at the time the game changes levels) to accomplish different effects. Application flexibility is not limited by any fixed or hardwired memory allocation.

Detailed Description Text (93):
Since all significant data structures are stored within common main memory 300, they can all be accessed by main processor 100 and other system elements. There is no hardware distinction between display images and source images. For example, main processor 100 can, if desired, directly access individual pixels within frame buffer 118. The scan conversion output of display processor 500 can be used as a texture for a texture mapping process. Image source data and scan converted image data can be interchanged and/or combined to accomplish special effects such as, for example,

warping scan-converted images into the viewpoint.

Detailed Description Text (94):
The shortcomings of a unified memory architecture (e.g., contention for access to the main memory 300 by different parts of the system) have been minimized through careful system design. Even though main memory 300 is accessed over a single narrow (9-bit-wide) bus 106 in this example, acceptable bandwidth has been provided by making the bus very fast (e.g., on the order of 240 MHz). Data caches are provided throughout the system 50 to make each sub-component more tolerant to waiting for main memory 300 to become available.

Detailed Description Text (118):
Display processor 500 in this example is a graphics display pipelined engine that renders a digital representation of a display image. It operates based on graphics display commands generated by the signal processor 400 and/or main processor 100. Display processor 500 includes, in addition to texture memory 502, a rasterizer 504, a texture unit 506, a color combiner 508, a blender 510 and a memory interface 512. Briefly, rasterizer 504 rasterizes polygon (e.g., triangle, and rectangle) geometric primitives to determine which pixels on the display screen 60 are within these primitives. The texture unit can apply texture maps stored within texture memory 502 onto textured areas defined by primitive edge equations solved by rasterizer 504. The color combiner 508 combines and interpolates between the texture color and a color associated with the graphic primitive. Blender 510 blends the resulting pixels with pixels in frame buffer 118 (the pixels in the frame buffer are accessed via memory interface 512) and is also involved in performing Z buffering (i.e., for hidden surface removal and anti-aliasing operations). Memory interface 512 performs read, modify and write operations for the individual pixels, and also has special modes for loading/copying texture memory 502, filling rectangles (fast clears), and copying multiple pixels from the texture memory 502 into the frame buffer 118. Memory interface 512 has one or more pixel caches to reduce the number of accesses to main memory 300.

Detailed Description Text (121):
Display processor 500 accesses main memory 300 using physical addresses to load its internal texture memory 502, read frame buffer 118 for blending, read the Z buffer 118B for depth comparison, to write to the Z-buffer and the frame buffer, and to read any graphics display commands stored in the main memory.

Detailed Description Text (125):
In this example, each of the coprocessor 200 sub-blocks shown has an associated direct memory access (DMA) circuit that allows it to independently address and access main memory 300. For example, signal processor DMA circuit 454, display processor DMA circuit 518, audio interface DMA circuit 1200, video interface DMA circuit 900, serial interface DMA circuit 1300, and parallel peripheral interface DMA circuit 1400 each allow their associated coprocessor sub-block to generate addresses on coprocessor address bus 214C and to communicate data via coprocessor data bus 214D (additionally, display processor 500 has a further memory interface block 512 for access to the main memory frame buffer 118 and texture data 116).

Detailed Description Text (130):
Vector unit 420 can perform the same operation on eight pairs of 16-bit operands in parallel simultaneously. This makes signal processor 400 especially suited for "sum of products" calculations such as those found in matrix multiplications, texture resampling, and audio digital signal processing such as, for example, digital audio synthesis and spatial and frequency filtering.

Detailed Description Text (147):
Texture definition/loading

Detailed Description Text (162):
FIG. 10 shows an example of a simplified graphics process performed by signal processor 400 based on a display list 110. In this simplified process, the display list 110 first commands signal processor 400 to set various attributes defining the overall graphical images that are to be rendered by the co-processor. Such attributes include, for example, shading, lighting, Z buffering, texture generation, fogging and culling (FIG. 10 block 612). The display list next commands signal processor 400 to define a modeling/viewing matrix and a projection matrix (FIG. 10, block 614). Once the appropriate matrices have been defined, the display list commands signal processor 400 to transform a set of vertices based on the modeling/viewing matrix and the projection matrix defined by block 614 and also based on the attributes set by block 612 (FIG. 10,

block 616). Finally, the display list commands signal processor 400 to generate a graphics display (e.g., triangle) command that directs display processor 500 to render a primitive based on the vertices generated by block 616 and the attributes set by block 612 (FIG. 10, block 618). Signal processor 400 may, in response to step 618, transfer the display processor command it has generated (or the address of the command, which the signal processor may store in its data memory 404 or in main memory 300) for access and execution by display processor 500.

Detailed Description Text (182):
Enable vertex shading or use primitive color to paint the polygon (default is vertex shading).

Detailed Description Text (188):
Enable z-buffer depth calculations.

Detailed Description Text (189):
G_TEXTURE_GEN

Detailed Description Text (190):
Enable automatic generation of the texture coordinates S & T. After transformations, a spherical mapping will be used to replace any S & T value originally given with the vertex.

Detailed Description Text (193):
G_TEXTURE_GEN_LINEAR

Detailed Description Text (194):
Enable linearization of the texture coordinates generated when G_TEXTURE_GEN is set. For example, this allows the use of a panoramic texture map when performing environment mapping.

Detailed Description Text (196):
Enable generation level of detail (LOD) value for mipmapped textures and texture-edge mode.

Detailed Description Text (204):
The ambient light is defined by a color: light.r, light.g, light.b (unsigned 8 bit integers) which should be set to the color of the ambient light multiplied by the color of the object which is to be drawn (If you are lighting a texture mapped object just use the color of the ambient light). (For ambient lights the light.x, light.y, and light.z fields are ignored). The ambient light cannot be turned off except by specifying a color of black in this example.

Detailed Description Text (205):
Directional lights are specified by a color: light.r, light.g, light.b (unsigned 8 bit integers) which, like the ambient light color, should be set to the color of the light source multiplied times the color of the object which is to be drawn. Directional lights also have a direction. The light.x, light.y, light.z fields (signed 8 bit fractions with 7 bits of fraction) indicates the direction from the object to light. There must be at least one directional light (if G_LIGHTING is enabled in G_SETGEOMETRYMODE command) turned on, but if its color is black it will have no effect on the scene.

Detailed Description Text (213):
This command turns texture mapping ON/OFF, provides texture coordinate scaling, and selects the tile number (within a tiled texture). Scale parameters are in the format of (0.16) and scale the texture parameters in vertex commands. Texture on/off turns on and off the texture coordinate processing in the geometry pipeline. Tile number corresponds to tiles chosen in the raster portion of the pipeline. The tile num also holds the maximum levels for level of detail (LOD) (mid-mapping). ##STR3##

Detailed Description Text (214):
This command is used for automatic texture coordinate generation. It is used to describe the orientation of the eye so that the signal processor 400 knows with respect to what to generate texture coordinates. The XYZ values (8 bit signed fractions with 7 bits of fraction) describe a vector in worldspace (the space between the MODELVIEW matrix and the PROJECTION matrix) which is perpendicular to the viewer's viewing direction and pointing towards the viewer's right.

Detailed Description Text (249):
FIG. 13B shows an example vertex data structure signal processor 400 uses to represent each of the vertices stored in vertex buffer 408. In this example, the transformed x, y, z, and w, values corresponding to the vertex are stored in double precision format, with the integer parts first followed by the fractional parts (fields 408(1)(a)-408(1)(h)). With vertex color (r, g, b, .alpha.) are stored in fields 408(1)(I)-408(1)(l), and vertex texture coordinates (s, t) are stored in fields 408(1)(m), 408(1)(n). Additionally, from this example, the vertex values in screen space coordinates (i.e., transformed and projected onto the viewing plane) are stored in fields 408(1)(o)-408(1)(t) (with the one/w value stored in double precision format). The screen coordinates are used by display processor 500 to draw polygons defined by the vertex. The transformed 3-dimensional coordinates are maintained in vertex buffer 408 for a clipping test. Since polygons (not vertices) are clipped, and since the vertices in vertex buffer 408 may be re-used for multiple polygons, these transformed 3D vertex values are stored for multiple possible clipping test to be performed. In addition, the vertex data structure 408(1) includes flags 408(1)(v) that signal processor 400 can use, for example, to specify clip test results (i.e., whether the vertex falls inside or outside of each of six different clip planes). The perspective projection factor stored in fields 408(1)(s), 408(1)(t) is retained for perspective correction operations performed by the display processor texture coordinate unit (explain below).

Detailed Description Text (251):
This command loads (n+1) points into the vector buffer beginning at location v0 in the vertex buffer. The segment id and address field are used to construct the main memory 300 address of the actual VTX structure. (see G_SEGMENT for more information). The number of vertices n, is encoded as "the number minus one", in order to allow a full 16 vertices to be represented in 4 bits. The length is the number of points times 16, the size of the VTX structure (in bytes). Vertex coordinates are 16-bit integers, the texture coordinates s and t are S10.5. The flag parameter is ignored in this example. A vertex either has a color or a normal (for shading). Colors are 8 bit unsigned numbers. Normals are 8 bit signed fractions (7 bits of fraction). (0x7f maps to +1.0, 0x81 maps to -1.0, and 0x0 maps to 0.0). Normal vectors must be normalized, i.e.,

Detailed Description Text (258):
Signal processor 400 next performs lighting calculations in order to "light" each of the vertices specified in the vertex command. System 50 supports a number of sophisticated real-time lighting effects, including ambient (uniform) lighting, diffuse (directional) lights, and specular highlights (using texture mapping). In order to perform lighting calculations in this example, signal processor 400 must first load an SP microcode 108 overlay to perform the lighting calculations. The G_SETGEOMETRYMODE command must have specified that lighting calculations are enabled, and the lights must have been defined by the G_NUM_LIGHTS command discussed above. The part of microcode 108 that performs the lighting calculations is not normally resident within signal processor 400, but is brought in through an overlay when lighting calls are made. This has performance implications for rendering scenes with some objects lighted and others colored statically. In this example, the lighting overlay overwrites the clipping microcode, so to achieve highest performance it is best to minimize or completely avoid clipped objects in lighted scenes.

Detailed Description Text (259):
To light an object, the vertices which make up the objects must have normals instead of colors specified. In this example, the normal consists of three signed 8-bit numbers representing the x, y and z components of the normal (see the G_VTX command format described above). Each component ranges in value from -128 to +127 in this example. The x component goes in the position of the red color of the vertex, the y into the green and the z into the blue. Alpha remains unchanged. The normal vector must be normalized, as discussed above.

Detailed Description Text (260):
Lighting can help achieve the effect of depth by altering the way objects appear as they change their orientation. Signal processor 400 in this example supports up to seven diffused lights in a scene. Each light has a direction and a color. Regardless of the orientation of the object and the viewer, each light will continue to shine in the same direction (relative to the open "world") until the light direction is changed. In addition, one ambient light provides uniform illumination. Shadows are not explicitly supported by signal processor 400 in this example.

Detailed Description Text (261):
As explained above, lighting information is passed to signal processor 400 in light

data structures. The number of diffuse lights can vary from 0 to 7. Variables with red, green and blue values represent the color of the light and take on values ranging from 0 to 255. The variables with the x, y, z suffixes represent the direction of the light. The convention is that the direction points toward the light. This means the light direction indicates the direction to the light and not the direction that the light is shining (for example, if the light is coming from the upper left of the world the direction might be x=-141, y=-141, z=0). To avoid any ambient light, the programmer must specify the ambient light is black (0, 0, 0,).

Detailed Description Text (263):
The lighting structures discussed above are used to provide color values for storing into vertex buffer fields 408(1)(i)-408(1)(l).

Detailed Description Text (264):
Texture Coordinate Scaling/Creation

Detailed Description Text (265):
Signal processor 400 next performs texture coordinate scaling and/or creation (FIG. 11, block 642). In this example, the operations performed by block 642 may be used to accomplish specular highlighting, reflection mapping and environment mapping. To render these effects, coprocessor 200 in this example uses a texture map of an image of the light or environment, and computes the texture coordinates s,t based on the angle from the viewpoint to the surface normal. This texture mapping technique avoids the need to calculate surface normals at each pixel to accomplish specular lighting. It would be too computationally intensive for system 50 in this example to perform such surface normal calculations at each pixel.

Detailed Description Text (266):
The specular highlight from most lights can be represented by a texture map defining a round dot with an exponential or Gaussian function representing the intensity distribution. If the scene contains highlights from other, oddly shaped lights such as fluorescent tubes or glowing swords, the difficulty in rendering is no greater provided a texture map of the highlight can be obtained.

Detailed Description Text (267):
Although display processor 500 performs texture mapping operations in this example, signal processor 400 performs texture coordinate transformations for each vertex when these effects are required. Activation or de-activation of the signal processor texture coordinate transformations is specified by a value within the G_SETGEOMETRYMODE Command (see above). In addition, the G_SETGEOMETRYMODE Command can specify linearization of the generated textured coordinates, e.g., to allow use of a panoramic texture map when performing environment mapping.

Detailed Description Text (268):
In this example, signal processor 400 texture coordinate generation utilizes the projection of the vertex normals in the x and y directions in screen space to derive the s and t indices respectively for referencing the texture. The angle between the viewpoint and the surface normal at each vertex is used to generate s, t. The normal projections are scaled to obtain the actual s and t values in this example. Signal processor 400 may map the vertices "behind" the point of view into 0, and may map positive projections into a scaled value.

Detailed Description Text (269):
In this example, texturing is activated using the G_TEXTURE command described above in the signal processor attribute command section. This command provides, among other things, scaling values for performing the texture coordinate mapping described above.

Detailed Description Text (270):
As explained above, the texture coordinate mapping performed by signal processor 400, in this example, also requires information specifying the orientation of the eye so that the angle between the vertex surface normal and the eye can be computed. The G_LOOKAT_X and the G_LOOKAT_Y commands supply the eye orientation for automatic texture coordinate generation performed by signal processor 400. The transformed texture coordinate values, if they are calculated, are stored by signal processor 400 in the vertex data structure at fields 408(1)(m), 408(1)(n). These texture coordinate values are provided to display processor 500 to perform acquired texture mapping using a texture specified by the G_TEXTURE command.

Detailed Description Text (271):

Since these effects use texture mapping, they cannot be used with objects which are otherwise texture mapped.

Detailed Description Text (279):
This command results in one triangle, using the vertices v0, v1, and v2 stored in the internal vertex buffer. The N field identifies which of the three vertices contains the normal of the face (for flat shading) or the color of the face (for flat shading).

Detailed Description Text (281):
This command generates one line, using the vertices v0 and v1 in the internal vertex buffer. The N field specifies which of the two vertices contain the color of the face (for flat shading).

Detailed Description Text (282):
Textured and filled rectangles require intervention by signal processor 400 and are thus a signal processor operation. The following is an example command format and associated function of a texture rectangle command:

Detailed Description Text (283):
These 3 commands draw a 2D rectangle with the current texture. The parameters x0, y0 specify the upper left corner of the rectangle; x1, y1 are the lower right corners. All coordinates are 12 bits. S and T are signed 10.5 bit numbers, and specify the upper left coordinate of s, t. DsDx and DtDy are signed 5.10 bit numbers, and specify change in s (t) per change in x (y) coordinate.

Detailed Description Text (284):
Signal processor 400 also in this example supports a G_TEXRECT_FLIP command that is identical to the G_TEXRECT command except that the texture is flipped so that the s coordinate changes in the y direction and the t coordinate changes in the x direction.

Detailed Description Text (285):
This command draws a 2D rectangle in the current fill color. The parameters x0, y0 specify the upper left comer of the rectangle; x1, y1 are the lower right comers. All coordinates are 12 bits.

Detailed Description Text (293):
The segmented addressing used by signal processor 400 in this example can be useful to facilitate double-buffered animation. For example, video game program 108 can keep two copies of certain display list fragments within main memory 300, with the same offsets in two different segments. Switching copies of them is as easy as swapping the segment pointers in signal processor 400. Another use is to group data and textures in one segment and to group static background geometry in another segment. Grouping data might help optimize memory caching in main processor 100. All data which contains embedded addresses must be preceded by the appropriate G_SEGMENT command that loads the signal processor 400 segment table with the proper base address.

Detailed Description Text (399):
Display processor 500 in this example rasterizes triangles and rectangles and produces high quality pixels that are textured, anti-aliased and z-buffered. FIG. 18 shows the overall processes performed by display processor 500. Display processor 500 receives graphics display commands that, for example, specify the vertices, color, texture, surface normal and other characteristics of graphics primitives to be rendered. In this example, display processor 500 can render lines, triangles, and rectangles. Typically, display processor 500 will receive the specifications for the primitives it is to render from signal processor 400, although it is also possible for main processor 100 to specify these commands directly to the display processor.

Detailed Description Text (400):
The first operation display processor 500 performs on an incoming primitive is to rasterize the primitive, i.e., to generate pixels that cover the interior of the primitive (FIG. 18, block 550). Rasterize block 550 generates various attributes (e.g., screen location, depth, RGBA color information, texture coordinates and other parameters, and a coverage value) for each pixel within the primitive. Rasterize block 550 outputs the texture coordinates and parameters to a texture block 552. Texture block 552 accesses texture information stored within texture memory 502, and applies ("maps") a texel (texture element) of a specified texture within the texture memory onto each pixel outputted by rasterized block 550. A color convert block 554 and a chroma keying block 556 further process the pixel value to provide a texture color to a color combine block 558.

Detailed Description Text (401):
Meanwhile, rasterize block 550 provides a primitive color (e.g., as a result of shading) for the same pixel to color combine block 558. Color combine block 558 combines these two colors to result in a single pixel color. This single pixel color output may have fog applied to it by block 560 (e.g., to create the effect of a smoke filled room, or the less extreme, natural effect of reducing color brilliance as an object moves further away from the viewer). The resulting pixel color value is then blended by a block 562 with a pixel value framebuffer 118 stores for the same screen coordinate location. An additional anti-alias/z-buffer operation 564 performs hidden surface removal (i.e., so closer opaque objects obscure objects further away), anti-aliasing (to remove jaggedness of primitive edges being approximated by a series of pixels), and cause the new pixel value to be written back into framebuffer 118.

Detailed Description Text (402):
The operations shown in FIG. 18 are performed for each pixel within each primitive to be rendered. Many primitives may define a single complex scene, and each primitive may contain hundreds or thousands of pixels. Thus, display processor 500 must process millions of pixels for each image to be displayed on color television set 58.

Detailed Description Text (405):
Because high speed operation is very important in rendering pixels, display processor 500 has been designed to operate as a "pipeline." Referring again to FIG. 18 "pipelining" means that the various steps shown in FIG. 18 can be performed in parallel for different pixels. For example, rasterize block 550 can provide a first pixel value to texture block 552, and then begin working on a next pixel value while the texture block is still working on the first pixel value. Similarly, rasterize block 550 may be many pixels ahead of the pixel that blend block 562 is working on.

Detailed Description Text (407):
FIG. 19B shows the two-cycle pipeline mode operation of display processor 500 in this example. In the FIG. 19B example, some of the operations shown in FIG. 18 are performed twice for each pixel. For example, the texture and color convert/filtering operations 552, 554 shown in FIG. 18 are repeated for each pixel; the color combine operation 558 is performed twice (once for the texture color output of one texture operation, and once for the texture color output of the other texture operation). Similarly, blend operation 562 shown in FIG. 18 is performed twice for each pixel.

Detailed Description Text (408):
Even though these various operations are performed twice, display processor 500 in this example does not contain duplicate hardware to perform the duplicated operations concurrently (duplicating such hardware would have increased cost and complexity). Therefore, in this example, display processor 500 duplicates an operation on a pixel by processing it with a particular circuit (e.g., a texture unit, a color combiner or a blender), and then using the same circuit again to perform the same type of operation again for the same pixel. This repetition slows down the pipeline by a factor of two (each pixel must "remain" at each stop in the pipeline for two cycles instead of one), but allows more complicated processing. For example, because the two-cycle-per-pixel mode can map two textures onto the same pixel, it is possible to do "trilinear" ("mipmapping") texture mapping. In addition, since in this example, display processor 500 uses the same blender hardware to perform both the fog operation 560 and the blend operation 562 (but cannot both blend and fog simultaneously), it is generally necessary to operate in the two-cycle-per-pixel mode to provide useful fog effects.

Detailed Description Text (411):
Display processor 500 also has a "fill" mode and a "copy" mode, each of which process four pixels per cycle. The fill mode is used to fill an area of framebuffer 118 with identical pixel values (e.g., for high performance clearing of the framebuffer or an area of it). The copy mode is used for high-performance image-to-image copying (e.g., from display processor texture memory 502 into a specified area of framebuffer 118). The copy mode provides a bit "blit" operation in addition to providing high performance copying in the other direction (i.e., from the framebuffer into the texture memory).

Detailed Description Text (415):
FIG. 20 shows an example architecture of display processor 500. In this example, display processor 500 includes a command unit 514 with associated RAM 516 and DMA controller 518; an "edge walker"/rasterizer 504; a RGBAZ pixel stepper 520; a color combiner/level interpreter 508, a blender/fogger 510, a ditherer 522, a coverage evaluator 524, a depth (z) comparator 526, a memory interface 512 and a texture unit

506. In this case, texture unit 506 includes, in addition to texture memory 502, texture steppers 528, a texture coordinate unit 530 and a texture filter unit 532.

Detailed Description Text (416):
Command unit 514 and DMA controller 518 connect to coprocessor main internal bus 214, and also connect to the signal processor 400 via a private "x" bus 218. Memory interface 512 is a special memory interface for use by display processor 500 primarily to access to the color framebuffer 118a and the z buffer 118b stored within main memory 300 (thus, display processor 500 has access to main memory 300 via memory interface 512 and also via coprocessor internal bus 214).

Detailed Description Text (418):
DMA controller 518 receives DMA commands from signal processor 400 or main processor 100 over bus 214. DMA controller 518 has a number of read/write registers shown in FIGS. 21A-21C that allow signal processor 400 and/or main processor 100 to specify a start and end address in SP data memory 404 or main memory 300 from which to read a string of graphics display commands (FIG. 21A shows a start address register 518A, and FIG. 21B shows an end address register 518B). DMA controller 518 reads data over main coprocessor bus 214 if registers 518a, 518b specify a main memory 300 address, and it reads data from the signal processor's data memory 404 over private "x bus" 214 if the registers 518a, 518b specify a data memory 404 address. DMA controller 518 also includes a further register (register 518C shown in FIG. 21C) that contains the current address DMA controller 518 is reading from. In this example, DMA controller 518 is uni-directional--that is, it can only write from bus 214 into RAM 516. Thus, DMA controller 518 is used in this example for reading from signal processor 400 or main memory 300. In this example, display processor 500 obtains data for its texture memory 502 by passing texture load commands to command unit 514 and using memory interface 512 to perform those commands.

Detailed Description Text (421):
For example, command unit 514 includes a status/command register 534 shown in FIG. 21D that acts as a status register when read by main processor 100 and acts as a command register when the main processor writes to it. When reading this register 534, main processor 100 can determine whether display processor 500 is occupied performing a DMA operation reading from signal processor data memory 404 (field 536(1); whether the display processor is stalled waiting for access to main memory 300 (field 536(2); whether the display processor pipeline is being flushed (field 536(3); whether the display processor graphics clock is started (field 536(4); whether texture memory 502 is busy (field 536(5); whether the display processor pipeline is busy (field 536(6); whether command unit 514 is busy (field 536(7); whether the command buffer RAM 516 is ready to accept new inputs (field 536(8); whether DMA controller 518 is busy (field 536(9); and whether the start and end addresses and registers 518a and 518b respectively valid (fields 536(10), 536(11). When writing to this same register 534, main processor 100 (or signal processor 400) can clear an X-bus DMA operation from the signal processor 400 (field 538(1); begin an X-bus DMA operation from signal processor data memory 404 (field 538(2); start or stop the display process (fields 538(3), 538(4); start or stop a pipeline flushing operation (fields 538(5), 538(6); clear a texture memory address counter 540 shown in FIG. 21H (field 538(7); clear a pipeline busy counter 542 shown in FIG. 21F (field 538(8); clear a command counter 544 used to index command buffer RAM 516 (field 538(9) (the counter 544 is shown in FIG. 21G); and clear a clock counter 546 (see FIG. 21E) used to count clock cycles (field 538 (10).

Detailed Description Text (422):
As mentioned above, the clock count, buffer count, pipeline count and texture memory count can all be read directly from registers 540-546 (see FIGS. 21E-21H). In addition, main processor 100 or signal processor 400 can read and control the BIST operation pertaining to texture memory 502 (see BIST status/control register 548 shown in FIG. 21I), and can also enable and control testing of memory interface 512 by manipulating mem span test registers 549(a), 549(b) and 549(c) shown in FIG. 21J.

Detailed Description Text (426):
Edgewalker 504 shown in FIG. 20 performs the rasterize process 550 shown in FIG. 18. In this example, edgewalker 504 receives the edge coefficients, shade coefficients, texture coefficients and z buffer coefficients specified in a "triangle command" (see FIGS. 46-118 specifying a particular primitive open line, triangle or rectangle), and outputs "span" values from which the following attributes for each pixel enclosed within the primitive can be derived:

Detailed Description Text (428):

z depth for z buffer purposes

Detailed Description Text (429):
RGBA color information

Detailed Description Text (430):
s/w, t, w, 1/w texture coordinates, level-of-detail for texture index, perspective correction, and mipmapping (these are commonly referred to s, t, w, 1)

Detailed Description Text (432):
Edgewalker 504 sends the parameters for a line of pixels across the primitive (a "span") to the pipeline hardware downstream for other computations. In particular, texture steppers 528 and RGBAZ steppers 520 receive the "span" information specified by edgewalker 504, and step sequentially along each pixel in the horizontal line (in the view plane coordinate system) of the "span" to derive the individual texture coordinates and RGBAZ values for each individual pixel in the span.

Detailed Description Text (434):
As mentioned above, steppers 520 produces color and alpha information for each pixel within the "span" defined by edgewalker 504. Similarly, texture steppers 528 produces texture coordinate values (s, t, w) for each pixel within the span. Steppers 520, 528 operate in a synchronized fashion so that texture unit 506 outputs a mapped texture value for a pixel to color combiner 58 at the same time that the RGBAZ steppers 520 output a color value for the same pixel based on primitive color, shading, lighting, etc.

Detailed Description Text (435):
Texture Unit

Detailed Description Text (436):
Texture unit 506 in this example takes the texture coordinates s, t, w and level-of-detail values for a pixel (as mentioned above, texture steppers 528 derive these values for each individual pixel based upon "span" information provided by edgewalker 504), and fetches appropriate texture information from onboard texture memory 502 for mapping onto the pixel. In this example, the four nearest texels to the screen pixel are fetched from texture memory 502, and these four texel values are used for mapping purposes. Video game program 108 can manipulate texture states such as texture image types and formats, how and where to load texture images, and texture sampling attributes.

Detailed Description Text (437):
Texture coordinate unit 530 computes appropriate texture coordinates for mapping texture stored within texture memory 502 onto the primitive being rendered. Since the 2-dimensional textures stored in texture memory 502 are square or rectangular images that must be mapped onto triangles of various sizes, the texture coordinate in 530 must select appropriate texels within the texture to map onto pixels in the primitive to avoid distorting the texture. See OpenGL Programming Guide at 278.

Detailed Description Text (438):
Texture coordinate unit 530 computes a mapping between the inputted pixel texture coordinates and four texels within the appropriate texture stored in texture memory 502. Texture coordinate unit 530 then addresses the texture memory 502 appropriately to retrieve these four texels. The four texel values are passed to the texture filter unit 532. Texture filter 532 takes the four texels retrieved from texture memory 502 and produces a simple bilinear-filtered texel. Texture filter 532 in this example can perform three types of filter operations: point sampling, box filtering, and bilinear interpolation. Point sampling selects the nearest texel to the screen pixel. In the special case where the screen pixel is always the center of four texels, the box filter can be used. In the case of the typical 3-D, arbitrarily rotated polygon, bilinear filtering is generally the best choice available. For hardware cost reduction, display processor texture filter unit 532 does not implement a true bilinear filter. Instead, it linearly interpolates the three nearest texels to produce the result pixels. This has a natural triangulation bias which is not noticeable in normal texture images but may be noticed in regular pattern images. This artifact can be eliminated by prefiltering the texture image with a wider filter. The type of filtering performed by texture filter unit 532 is set using parameters in the "set modes" display command (see FIGS. 46-118).

Detailed Description Text (439):

Texture Memory 502

Detailed Description Text (440):
Display processor 500 treats texture memory 502 as a general-purpose texture memory. In this example, texture memory 502 is divided into four simultaneously accessible banks, giving output of four texels per clock cycle. Video game program 58 can load varying-sized textures with different formats anywhere in the texture memory 502. Texture coordinate unit 530 maintains eight texture tile descriptors that describe the location of texture images within texture memory 502, the format of each texture, and its sampling parameters. This allows display processor 500 to access as many as eight different texture tiles at a time (more than eight texture tiles can be loaded into the texture memory, but only eight tiles are accessible at any time).

Detailed Description Text (441):
FIG. 22 shows an example of the texture tile descriptors and their relationship to texture tiles stored in texture memory 502. In this particular example shown in FIG. 22, eight different texture tiles 802 are stored within texture memory 502. Each texture tile 802 has an associated texture tile descriptor block 804 (as discussed above, display processor 500 maintains up to eight descriptors 804 corresponding to eight texture tiles stored within texture memory 502). The texture descriptors contain information specified by a "set tile" command (see FIGS. 46-118). For example, these texture tile descriptors specify the image data format (RGBA, YUV, color index mode, etc.), the size of each pixel/texel color element (four, eight, sixteen, thirty-two bits), the size of the tile line in 64-bit words, the starting address of the tile in texture memory 502, a palette number for 4-bit color indexed texels, clamp and mirror enables for each of the S and T directions, masks for wrapping/mirroring in each of S and T directions, level of detail shifts for each of S and T addresses. These descriptors 804 are used by texture coordinate unit 530 to calculate addresses of texels within the texture memory 502.

Detailed Description Text (442):
Texture Coordinate Unit

Detailed Description Text (443):
FIG. 23 shows a more detailed example of the processing performed by texture coordinate unit 530. FIG. 23 shows the various tile descriptors 804 being applied as inputs to texture coordinate unit 530. FIG. 23 also shows that texture coordinate unit 530 receives the primitive tile/level/texture coordinates for the current pixel from texture steppers 528. Texture coordinate unit 530 additionally receives mode control signals from command unit 514 based, for example, on the "set other mode" and "set texture image" commands (see FIGS. 46-118). Based on all of this input information, texture coordinate unit 530 calculates which tile descriptor 804 to use for this primitive, and converts the inputted texture image coordinates to tile-relative coordinates which the texture coordinate unit wraps, mirrors and/or clamps as specified by the tile descriptor 804. Texture coordinate unit 530 then generates an offset into texture memory 502 based on these tile coordinates. The texture coordinate unit 530 in this example can address 2.times.2 regions of texels in one or two cycle mode, or 4.times.1 regions in copy mode. Texture coordinate unit 530 also generates S/T/L fraction values that are used to bi-linearly or tri-linearly interpolate the texels.

Detailed Description Text (444):
FIG. 24 is a detailed diagram of texture coordinate unit 530 and texture memory unit 502. As shown in FIG. 24, the incoming s, t, w texture coordinates are inputted into a perspective correction block 566 which provides a perspective correction based on w when perspective correction is enabled. The perspective-corrected s, t values are then provided to a level-of-detail or precision shift block 568 which shifts the texture coordinates after perspective divide (e.g., for MIP mapping and possibly for precision reasons). A block 570 then converts the shifted texture coordinates to tile coordinates, providing fractional values to the texture filter unit 532. These tile coordinate values are then clamped, wrapped and/or mirrored by block 572 based on the current texture mode parameters of display processor 500. Meanwhile, the perspective-corrected texture coordinates provided by perspective correction block 566 are also provided to a level of detail block 574 which, when level of detail calculations are enabled, calculates a tile descriptor index into a tile descriptor memory 576 and also calculates a level of detail fractional value for interpolation by the color combiner 508. The tile descriptors 804 are stored in tile descriptor memory 576, and are retrieved and outputted to a memory conversion block 578 which conversion block also receives the adjusted texture coordinate values of block 572. Address conversion block 578 converts the adjusted texture coordinate values into texture

memory unit addresses based on current tile size, format and other parameters as specified by the tile descriptor 804. Address conversion block 578 outputs the texel address to texture memory unit 502. The texture memory unit 502 also receives additional parameters which are used, for example, if the texture is color indexed. Texture memory unit 502 outputs four texel values to texture filter unit 532 for filtering as discussed above.

Detailed Description Text (445):
Texture Memory Loading

Detailed Description Text (446):
Texture memory unit 502 includes a four kilobyte random access memory onboard coprocessor 200. Because texturing requires a large amount of random accesses with consistent access time, it is impractical to texture directly from main memory 300 in this example. The approach taken is to cache up to four kilobytes of an image in on-chip, high-speed texture memory 502. All primitives can be textured using the contents of texture memory 502.

Detailed Description Text (447):
In order to use texture memory 502, video game program 108 must load a texture tile into the texture memory and then load the associated descriptor 804 into tile descriptor 576. The "load tile" command (see FIGS. 46-118) is used to load a tile into texture memory 502, and a "set tile" and "set tile size" command are used to load corresponding tile descriptor blocks 804 into tile descriptor memory 576. In addition, a "Load Tlut" command (see FIGS. 46-118) can be used to load a color lookup table into texture memory 502 for use by color indexed textures.

Detailed Description Text (448):
Physically, texture memory 502 is organized in four banks, each comprising 256 16-bit wide words, each bank having a low half and a high half. This organization can be used to store 4-bit textures (twenty texels per row), 8-bit textures (ten texels per row), 16-bit textures (six texels per row), 16-bit YUV textures (twelve texels per row), and 32-bit textures (six texels per row). In addition, texture unit 506 in this example supports a color-indexed texture mode in which the high half of texture memory 502 is used to store a color lookup table and the low half of the texture memory is used to store 4-bit or 8-bit color indexed textures. This organization is shown in FIG. 25. In this FIG. 25 example, a color indexed texture tile 580 is stored in a low half 502(L) of texture memory 502, and a corresponding color lookup table 582 is stored in the upper half 502(H) of the texture memory.

Detailed Description Text (449):
FIG. 26 shows a more detailed depiction of a particular texture memory color indexed mode, in which the color lookup table 582 is divided into four palette banks 584 or tables, each having, for example, sixteen entries, each entry being 16-bits wide. The color lookup table may represent color in 16-bit RGBA format, or in 16-bit IA format. Since four texels are addressed simultaneously, there are four (usually identical) lookup tables 484 stored in the upper half of texture memory 502. As mentioned above, these lookup tables are loaded using the "load Tlut" command shown in FIGS. 46-118.

Detailed Description Text (450):
Display processor 500 supports another color-indexed texture mode in which each texel in the lower half of texture memory 502 comprises eight bits--and therefore can directly access any one of the 256 locations in the upper half 502(H) of texture memory 502. Thus, 8-bit color-indexed textures do not use the palette number of the tile, since they address the whole 256-element lookup table directly. It is not necessary to use the entire upper half of texture memory 502 for a lookup table when using 8-bit color-indexed textures. For example, if less than eight of the bits of the 8-bit color-indexed texture tile is being used for color lookup, only a portion of color memory upper half 502(H) is required to store the lookup table--and the remainder of the upper half of the texture memory 502 might thus be used for storing a non-color-indexed texture such as a 4-bit I texture (see FIG. 25). Similarly, even when color-indexed texture 580 is stored in the lower half 502(L) of texture memory 502, it is possible to also store non-color-indexed textures in the lower half as well. Thus, color-indexed textures and non-color-indexed textures can be co-resident in texture memory 502.

Detailed Description Text (451):
The following texture formats and sizes are supported by texture memory 502 and texture coordinate unit 530:

Detailed Description Text (452):
In this example, texture unit 506 will, unless explicitly told otherwise, change a tile descriptor 804 or a texture tile 802 immediately upon loading--even if it is still being used for texture mapping of a previous primitive. Texture loads after primitive rendering should be preceded by a "sync load" command and tile descriptor attribute changes should be preceded by a "sync tile" command to ensure that the texture tile and tile descriptor state of texture unit 506 does not change before the last primitive is completely finished processing (see FIGS. 46-118 for example formats and functions of these commands).

Detailed Description Text (453):
As mentioned above in connection with the signal processor 400, two special commands ("texture rectangle" and "texture rectangle flip") can be used to map a texture onto a rectangle primitive (see FIGS. 46-118). It is possible to use the "texture rectangle" command to copy an image from texture memory 502 into frame buffer 118, for example. See FIGS. 46-118.

Detailed Description Text (454):
Color Combiner

Detailed Description Text (455):
Referring once again to FIG. 20, color combiner 508 combines texels outputted by texture unit 506 with stepped RGBA pixel values outputted by RGBAZ steppers 520. Color combiner 508 can take two color values from many sources and linearly interpolate between them. The color combiner 508 performs

Detailed Description Text (456):
the equation: Here, A, B, C and D can come from many different sources (note that if D=B, then color combiner 508 performs simple linear interpolation).

Detailed Description Text (457):
FIG. 27 shows possible input selection of a general purpose linear interpolator color combiner 508 for RGB and Alpha color combination in this example. As can be seen in FIG. 27, only some of the inputs in the lefthand column come from texture unit 506 or RBGAZ steppers 520. The rest of the inputs are derived from color combiner 508 internal state that can be programmed by sending commands to display processor 500. As discussed above, the "combined color" and "combined Alpha" values provided to color combiner 508 are obtained from the RGBAZ steppers 520, and the texel color and texture Alpha are obtained from texture unit 506 (two texel colors and corresponding Alpha values are shown since in two-cycle-per-pixel mode two texels will be provided by texture unit 506 for purposes of mipmapping for example). Additionally, the level of detail fractional input is obtained from FIG. 24 block 574, and the primitive level of detail value along with the primitive color and primitive Alpha value may be obtained from a "set primitive color" command sent to display processor 500 (see FIGS. 46-118) (the primitive color value/alpha/level of detail fraction value can be used to set a constant polygon face color). Similarly, a shade color and associated Alpha value may be obtained from a "shade coefficient" command (see FIGS. 46-118), and an environment color and associated Alpha value may be obtained from a "set environment color" command (see FIGS. 46-118) (the environment color/alpha value described above can be used to represent the ambient color of the environment). Two kinds of "set key" commands (one for green/blue, the other for red) are used for green/blue color keying and red color keying respectively--these supplying the appropriate key:center and key:scale inputs to color combiner 508 (see FIGS. 46-118). Both the primitive and environment values are programmable and thus can be used as general linear interpolation sources.

Detailed Description Text (458):
Convert K4 and K5 Inputs to color combiner 508 are specified in this example by the "set convert" command (see FIGS. 46-118) that adjust red color coordinates after conversion of texel values from YUV to RGB format (the remainder of the conversion process responsive to this set convert command being performed within texture filter unit 532).

Detailed Description Text (459):
FIG. 28 shows a portion of color combiner 508 used for combining the alpha values shown as inputs in FIG. 27. For both the RGB color combine in alpha color combine operations performed by color combiner 508, there are two modes, one for each of the two possible pipeline modes one cycle-per-pixel, and two cycles-per-pixel). In the two-cycle mode, color combiner 508 can perform two linear interpolation arithmetic computations.

Typically, the second cycle is used to perform texture and shading color modulation (i.e., the operations color combiner 508 are typically used for exclusively in the one-cycle mode), and the first cycle can be used for another linear interpolation calculation (e.g., level of detail interpolation between two bi-linear filtered texels from two mipmap tiles). Color combiner 508 also performs the "alpha fix-up" operation shown in FIG. 29 in this example (see "set key GB" command in FIGS. 46-118).

Detailed Description Text (461):
As discussed above, blender 510 takes the combined pixel value provided by color combiner 508 can blends them against the frame buffer 118 pixels. Transparency is accomplished by blending against the frame buffer color pixels. Polygon edge antialias is performed, in part, by blender 510 using conditional color blending based on depth (z) range. The blender 510 can also perform fog operations in two-cycle mode.

Detailed Description Text (462):
Blender 510 can perform different conditional color-blending and z buffer updating, and therefore can handle all of the various types of surfaces shown in FIG. 30 (i.e., opaque surfaces, decal surfaces, transparent surfaces, and inter-penetrating surfaces).

Detailed Description Text (463):
An important feature of blender 510 is its participation in the antialias process. Blender 510 conditionally blends or writes pixels into frame buffer 118A based on depth range (see FIG. 33 which shows example z buffer formats including a "dz" depth-range field). See U.S. Patent Application Ser. No. 08/062,283 of Akeley et al, entitled "System and Method For Merging Pixel Fragments Based On Depth Range Values", filed concurrently herewith.

Detailed Description Text (464):
In this example, video interface 210 applies a spatial filter at frame buffer read-out time to account for surrounding background colors to produce antialias silhouette edges. The antialiasing scheme requires ordered rendering sorted by surface or line types. Here is the rendering order and surface/line types for z buffer antialiasing mode:

Detailed Description Text (469):
These can be rendered in any order, but proper depth order gives proper transparency.

Detailed Description Text (471):
Blender 510 has two internal color registers: fog color and blend color. These values are programmable using the "set fog color" and "set blend color" commands, respectively (see FIGS. 46-118). These values can be used for geometry with constant fog or transparency.

Detailed Description Text (472):
Blender 510 can compare the incoming pixel alpha value with a programmable alpha source to conditionally update frame buffer 118A. This feature can allow complex, outlined, billboard type objects, for example. Besides thresholding against a value, blender 510 in this example can also compare against a dithered value to give a randomized particle effect. See "set other modes" command (FIGS. 46-118). Blender 510 can also perform fog operations, either in 1-cycle or 2-cycle mode. Blender 510 uses the stepped z value as a fog coefficient for fog and pipeline color blending.

Detailed Description Text (473):
FIG. 31 shows an example of the overall operations performed by blender 510 in this example. In this particular example, blender 510 can be operated in a mode in which a coverage value produced by coverage evaluator 524 can be used to specify the amount of blending. Coverage evaluator 524 compares the coverage value of the current pixel (provided by edge walker 504) to stored coverage value within frame buffer 118A. As shown in FIG. 32 (a depiction of the format of the color information stored for each pixel within color frame buffer 118A), the color of a pixel is represented by 5-bits each of red, green, and blue data and by a 3-bit "coverage" value. This "coverage" value can be used as-is, or multiplied by an alpha value for use as pixel alpha and/or coverage (see "set other modes" command in FIGS. 46-118). The "coverage" value nominally specifies how much of a pixel is covered by a particular surface. Thus, the coverage value outputted by edge walker 504 will be 1 for pixels lying entirely within the interior of a primitive, and some value less than 1 for pixels on the edge of the primitive. In this example, blender 510 uses the coverage value for antialiasing. At the time blender 510 blends a primitive edge, it does not know whether the primitive

edge is internal to an object formed from multiple primitives or whether the edge is at the outer edge of a represented object. To solve this problem in this example, final blending of opaque edge values is postponed until display time, when the video interface 210 reads out frame buffer 118A for display purposes. Video interface 210 uses this coverage value to interpolate between the pixel color and the colors of neighboring pixels in the frame buffer 118A. In order to accomplish this antialiasing at display time, blender 510 must maintain the coverage value for each pixel within frame buffer 118a, thereby allowing video interface 210 to later determine whether a particular pixel is a silhouette edge or an internal edge of a multi-polygon object.

Detailed Description Text (475):
Memory interface 512 provides an interface between display processor 500 and main memory 300. Memory interface 512 is primarily used during normal display processor 500 operations to access the color frame buffer 118a and the Z buffer 118b. Color frame buffer 118a stores a color value for each pixel on color television screen 60. The pixel format is shown in FIG. 32. Z buffer 118b stores a depth value and a depth range value for each color pixel value stored in color frame buffer 118a. An example format for z buffer values is shown in FIG. 33. The Z buffer 118b is used primarily by blender 510 to determine whether a newly rendered primitive is in front of or behind a previously rendered primitive (thereby providing hidden surface removal). The "DZ" depth range value shown in FIG. 33 may be used to help ascertain whether adjacent texels are part of the same object surface.

Detailed Description Text (478):
Depth comparator 526 operates in conjunction with z buffer 118b to remove hidden surfaces and to insure the transparent values are blended properly. Depth comparator 526 compares the z or depth value of the current pixel with the z value currently residing in z buffer 118a for that screen location. At the beginning of the rendering of a new frame, all locations in z buffer 118b are preferably initialized to maximum distance from the viewer (thus, any object will be open "in front of" this initialized value). Generally, each time display processor 500 is to blend a new pixel into frame buffer 118a, depth comparator 526 compares the depth of the current pixel with the depth residing in that location of z buffer 118b. If the old z buffer value indicates that the previously written pixel is "closer" to the viewer than is the new pixel, the new pixel is discarded (at least for opaque values) and is not written into the frame buffer--thus accomplishing hidden surface removal. If the new pixel is "closer" to the old pixel as indicated by depth comparator 526, then the new pixel value (at least for opaque pixels) may replace the old pixel value in frame buffer 118a--and the corresponding value in z buffer 118b is similarly updated with the z location of the new pixel (see FIG. 33A). Transparency blending may be accomplished by blending without updating the z buffer value--but nevertheless reading it first and not blending if the transparent pixel is "behind" an opaque pixel.

Detailed Description Text (481):
Video interface 210 in this example works in either NTSC or PAL mode, and can display 15-bit or a 24-bit color pixels with or without filtering at both high and low resolutions. The video interface 210 can also scale up a smaller image to fill the screen. The video interface 210 provides 28 different video modes plus additional special features.

Detailed Description Text (482):
Video interface 210 reads color frame buffer 118a in synchronization with the electron beam scanning the color television screen 60, and provides RGB values for each pixel in digital form to video DAC 144 for conversion into analog video levels in this example. Video interface 210 performs a blending function for opacity values based on coverage (thereby providing an antialiasing function), and also performs a back-filtering operation to remove some of the noise introduced by screen-based dithering.

Detailed Description Text (484):
DMA controller 900 is connected coprocessor bus 214. DMA controller 900 reads color frame buffer 118a beginning at an "origin" address in the main memory specified by main process 100 (see FIG. 35B). DMA controller 900 sequentially reads the pixel color and coverage values (see FIG. 32) from frame buffer 118a in synchronism with the line scanning operations of television 58. The pixel values read by DMA controller 900 are processed by the remainder of video interface 210 and are outputted to video DAC 144 for conversion into an analog composite video signal NTSC or PAL format in this example.

Detailed Description Text (485):

DMA controller 900 in this example provides the color/coverage values it has read from main memory frame buffer 118a, to a RAM buffer 902 for temporary storage. In this example, buffer 902 does not store the pixel color values corresponding to an entire line of television video. Instead, buffer 902 stores a plurality of blocks of pixel data, each block corresponding to a portion of a line of video. Buffer 902 provides "double buffering," i.e., it has sufficient buffers to make some line portions available to filters 906 while other buffers are being written by DMA controller 900.

Detailed Description Text (487):
Each of these blocks of pixel values is stored in buffer 902. Filters 906a, 906b perform a filtering/anti-aliasing operation based on coverage value to interpolate the current line's pixel values with neighboring pixel values (i.e., pixel values that are adjacent with respect to the displayed position on color television screen 60). The anti-aliasing filtering operations performed by filters 906a, 906b are as described in co-pending U.S. patent application Ser. No. 08/539,956 of Van Hook et al, entitled "Antialiasing of Silhouette Edges," filed on Oct. 6, 1995. Briefly, a three-scan-line high neighborhood is color weighted by coverage value in a blending process performed by filter 906. This filtering operation results in smoother, less jagged lines at surface edges by using the pixel coverage value retained in frame buffer 118a (which coverage value indicates what percentage of the pixel is covered by a polygon) to adjust the contribution of that pixel value relative to the contributions of neighboring pixel values in a blending process to produce the current pixel value. "Divot" error correction blocks 908a, 908b correct the outputs of anti-alias filters 906a, 906b for slight artifacts introduced by the anti-aliasing process. In particular, for any pixels on or adjacent to a silhouette edge, the error correction blocks 908 take the median of three adjacent pixels as the color to be displayed in place of the center pixel. This error correction can be enabled or disabled under software control (see FIG. 35A), and a video game programmer may wish to disable the error correction since it interacts poorly with decal line rendering modes.

Detailed Description Text (491):
The output of horizontal interpolator 912 is provided to a Gamma correction circuit 916 that converts linear RGB intensity into non-linear intensity values suitable for composite video generation for the gamma non-linearity of TV monitors. This amounts to taking a square root of the linear color space. The TV monitor effectively raises these color values to a power of 2.2 or 2.4. A "random" function block 914 introduces additional bits of resolution to each of the R, G and B color values in order to "de-dither" (i.e., to compensate for the bit truncation performed by display processor dithering block 522). As shown in FIG. 32, one example frame buffer 118 color pixel format in this example provides only five bits of resolution of each R, G and B to conserve storage space within main memory 300. Display processor dithering block 522 may truncate 8-bit RGB color values provided by blender 510 to provide the compressed representation shown in FIG. 32. Block 914 can reverse this truncation process to decompress the RGB values to provide 256 different display color levels for each R, G and B. See U.S. Pat. No. 5,699,079. This dither filter operation can be turned on and off under software control (see FIG. 35A).

Detailed Description Text (495):
Type field 952a specifies pixel data size as blank (no data, no sync), the format shown in FIG. 32 (5 bits each of ROB and a 3-bit coverage value), or 8/8/8/8 (32-bit color value and 8 bits of coverage);

Detailed Description Text (509):
FIG. 35F shows the vertical interface timing register 962 which main processor 100 can write to to specify horizontal sync pulse width, color burst width, vertical sync pulse width, and color burst start timing.

Detailed Description Text (514):
The vertical interfaced vertical burst register 972 shown in FIG. 35L specifies color burst start and end timing.

Detailed Description Text (515):
The timing parameters programmable into registers 962-972 can be used to provide compatibility with different kinds of television sets 58. For example, most television sets 58 in the United States use a composite video format known as NTSC, whereas most European television sets use a composite video format known as PAL. These formats differ in terms of their detailed timing parameters (e.g., vertical blanking integral width and location within the signal pattern, horizontal synchronization pulse width, color burst signal pulse width, etc.). Because registers 962-972 control these

composite video timing parameters and are programmable by software executing on main processor 100, a programmer of video game 108 can make her program NTSC compatible, PAL compatible, or both (as selected by a user) by including appropriate instructions within the video game program that write appropriate values to registers 962-972. Thus, in this example, coprocessor 200 is compatible with NTSC-standard television sets 58 with, PAL standard compatible television sets--and even with video formats other than these within a range as specified by the contents of registers 962-972.

Detailed Description Paragraph Table (7):
G_TEXRECT command x0 y0 x1 y1 command 0x000000 S (top left texture coord) T(top left texture coord) command 0x000000 DsDx DtDy

Detailed Description Paragraph Table (24):
Block Functionality Display Processor Pipeline Block Functionality in One-Cycle Mode Rasterize 550 Generates pixel and its attribute covered by the interior of the primitive. Texture 552 Generates 4 texels nearest to this pixel in a texture map. Filter Texture 554 Bilinear filters 4 texels into 1 texel, OR performs step 1 of YUV-to-RGB conversion. Combine 558 Combines various colors into a single color, OR performs step2 of YUV-to-RGB conversion. Blend 562 Blends the pixel with framebuffer memory pixel, OR fogs the pixel for writing to framebuffer. Framebuffer 563 Fetches and writes pixels (color and z) from and to the framebuffer memory. Display Processor Pipeline Block Functionality in Two-Cycle Mode Rasterize 550 Generates pixel and its attribute covered by the interior of the primitive. Texture 552a Generates 4 texels nearest to this pixel in a texture map. This can be level X of a mipmap. Texture 552b Generates 4 texels nearest to this pixel in a texture map. This can be level X + 1 of a mipmap. Filter Texture 554a Bilinear; filters 4 texels into 1 texel. Filter Texture 554b Bilinear; filters 4 texels into 1 texel. Combine 558a Combines various colors into a single color, OR linear interpolates the 2 bilinear filtered texels from 2 adjacent levels of a mipmap, OR performs step 2 of YUV-to-RGB conversion. Combine 558b Combines various colors into a single color, OR chroma keying. Block Functionality Blend 562a Combines fog color with resultant CC 1 color. Blend 562b Blends the pipeline pixels with framebuffer memory pixels. Framebuffer 563a Read/modify/write color memory; and Framebuffer 563b Read/modify/write Z memory.

Detailed Description Paragraph Table (25):
Texture Format and Sizes Type 4-bit 8-bit 16-bit 32-bit RGBA X X YUV X Color Index X X Intensity Alpha (IA) X X X Intensity (I) X X

Other Reference Publication (1):
Johnson, Matthew, A Fixed-Point DSP For Graphics Engines, Aug. 1989, IEEE, Los Alamitos, CA, pp. 63-77.

CLAIMS:

1. In a video game system including a microcomputer, a graphics and audio coprocessor, and a randomly accessible memory capable of at least in part operating as a frame buffer, said graphics and audio coprocessor comprising:

a signal processor shared between graphics processing and audio processing, said signal processor including a scalar processing unit, a vector processing unit, an instruction memory and a data memory having a task list including both graphics tasks and audio tasks, the scalar processing unit and the vector processing unit executing instructions from the instruction memory to perform the graphics tasks and audio tasks from the task list to provide graphics and audio data;

a display processor, coupled to said signal processor, the display processor including a rasterizer circuit, a texture unit and associated texture memory, a color combiner, a blender circuit for, in use, blending the color combiner output with contents of the frame buffer, and a memory interface circuit for, in use, being coupled to the random access memory;

an input/output circuit for, in use, being coupled to the microcomputer, a video display, an audio reproducer, an external program memory, and at least one humanly manipulable input device; and

a main bus and a private bus for connecting said signal processor and said display processor.

5. A 3-D video graphics system for generating 3D graphics for display on a home color

television set and for generating sound for reproduction by the home color television set, the system responding interactively to game controller user inputs based at least in part on a stored program provided at least in part by a portable replaceable memory device, the system including a high speed programmable 3D graphics and audio coprocessor comprising a signal processor including a scalar processing unit for performing scalar graphics computations, a vector processing unit for performing parallel vector graphics calculations, an instruction memory for holding instructions and a data memory for holding data, the scalar processing unit and the vector processing unit both executing instructions from the instruction memory to provide graphics and audio data; a display processor responsive to graphics data provided by said signal processor for rendering geometric shapes to create a display image; and a main bus and a private bus for connecting said signal processor and said display processor.

38. A coprocessor as in claim 21, wherein said display processor comprises a rasterizer circuit, a texture unit and associated texture memory, a color combiner, a blender circuit for, in use, blending the color combiner output with contents of said frame buffer, and a memory interface circuit for, in use, being coupled to a random access memory capable of at least in part operating as a frame buffer.

WEST

Generate Collection Print

L4: Entry 20 of 25 File: USPT Aug 15, 2000

DOCUMENT-IDENTIFIER: US 6104417 A
TITLE: Unified memory computer architecture with dynamic graphics memory allocation

Abstract Text (1):
A computer system provides dynamic memory allocation for graphics. The computer system includes a memory controller, a unified system memory, and memory clients each having access to the system memory via the memory controller. Memory clients can include a graphics rendering engine, a CPU, an image processor, a data compression/expansion device, an input/output device, a graphics back end device. The computer system provides read/write access to the unified system memory, through the memory controller, for each of the memory clients. Translation hardware is included for mapping virtual addresses of pixel buffers to physical memory locations in the unified system memory. Pixel buffers are dynamically allocated as tiles of physically contiguous memory. Translation hardware is implemented in each of the computational devices, which are included as memory clients in the computer system, including primarily the rendering engine.

Brief Summary Text (7):
One problem with the prior art computer graphics system 100 is the cost of high performance peripheral dedicated memory systems such as the dedicated graphics memory unit 104 and dedicated image processor memory 105. Another problem with the prior art computer graphics system 100 is the cost of high performance interconnects for multiple memory systems. Another problem with the prior art computer graphics system 100 is that the above discussed transfers of data between memory units require time and processing resources.

Brief Summary Text (8):
Thus, what is needed is a computer system architecture with a single unified memory system which can be shared by multiple processors in the computer system without transferring data between multiple dedicated memory units.

Brief Summary Text (10):
The present invention pertains to a computer system providing dynamic memory allocation for graphics. The computer system includes a memory controller, a unified system memory, and memory clients each having access to the system memory via the memory controller. Memory clients can include a graphics rendering engine, a central processing unit (CPU), an image processor, a data compression/expansion device, an input/output device, and a graphics back end device. In a preferred embodiment, the rendering engine and the memory controller are implemented on a first integrated circuit (first IC) and the image processor and the data compression/expansion are implemented on a second IC. The computer system provides read/write access to the unified system memory, through the memory controller, for each of the memory clients. Translation hardware is included for mapping virtual addresses of pixel buffers to physical memory locations in the unified system memory. Pixel buffers are dynamically allocated as tiles of physically contiguous memory. Translation hardware, for mapping the virtual addresses of pixel buffers to physical memory locations in the unified system memory, is implemented in each of the computational devices which are included as memory clients in the computer system.

Brief Summary Text (11):
In a preferred embodiment, the unified system memory is implemented using synchronous DRAM. Also in the preferred embodiment, tiles are comprised of 64 kilobytes of physically contiguous memory arranged as 128 rows of 128 pixels wherein each pixel is a 4 byte pixel. However, the present invention is also well suited to using tiles of other sizes. Also in the preferred embodiment, the dynamically allocated pixel buffers are comprised of n.sup.2 tiles where n is an integer.

Brief Summary Text (12):
The computer system of the present invention provides functional advantages for graphical display and image processing. There are no dedicated memory units in the computer system of the present invention aside from the unified system memory. Therefore, it is not necessary to transfer data from one dedicated memory unit to another when a peripheral processor is called upon to process data generated by the CPU or by another peripheral device.

Drawing Description Text (4):
FIG. 2A is a circuit block diagram of an exemplary unified system memory computer architecture according to the present invention.

Drawing Description Text (5):
FIG. 2B is an internal circuit block diagram of a graphics rendering and memory controller IC including a memory controller (MC)and a graphics rendering engine integrated therein.

Drawing Description Text (11):
FIG. 5 is a timing diagram for memory client requests issued to the unified system memory according to the present invention.

Drawing Description Text (14):
FIG. 8 is a timing diagram for an exemplary write to a new page performed by the unified system memory according to the present invention.

Drawing Description Text (15):
FIG. 9 is a timing diagram for an exemplary read to a new page performed by the unified system memory according to the present invention.

Detailed Description Text (5):
With reference to FIG. 2A, a computer system 200, according to the present invention, is shown. Computer system 200 includes a unified system memory 204 which is shared by various memory system clients including a CPU 206, a graphics rendering engine 208, an input/output IC 210, a graphics back end IC 212, an image processor 214, data compression/expansion device 215 and a memory controller 204.

Detailed Description Text (6):
With reference to FIG. 2B, an exemplary computer system 201, according to the present invention, is shown. Computer system 201 includes the unified system memory 202 which is shared by various memory system clients including the CPU 206, the input/output IC 210, the graphics back end IC 212, an image processing and compression and expansion IC 216, and a graphics rendering and memory controller IC 218. The image processing and compression and expansion IC 216 includes the image processor 214, and a data compression and expansion unit 215. GRMC IC 218 includes the graphics rendering engine (rendering engine) 208 and the memory controller 204 integrated therein. The graphics rendering and memory controller IC 218 is coupled to unified system memory 202 via a high bandwidth memory data bus (HBWMD BUS) 225. In a preferred embodiment of the present invention, HBWMD BUS 225 includes a demultiplexer (SD-MUX) 220, a first BUS 222 coupled between the graphics rendering and memory controller IC 218 and SD-MUX 220, and a second bus 224 coupled between SD-MUX 220 and unified system memory 202. In the preferred embodiment of the present invention, BUS 222 includes 144 lines cycled at 133 MHz and BUS 224 includes 288 lines cycled at 66 MHz. SD-MUX 220 demultiplexes the 144 lines of BUS 222, which are cycled at 133 MHz, to double the number of lines, 288, of BUS 224, which are cycled at half the frequency, 66 MHz. CPU 206 is coupled to the graphics rendering and memory controller IC 218 by a third bus 226. In the preferred embodiment of the present invention, BUS 226 is 64 bits wide and carries signals cycled at 100 MHz. The image processing and compression and expansion IC 216 is coupled to BUS 226, by a third bus 228. In the preferred embodiment of the present invention, BUS 228 is 64 bits wide and carries signals cycled at 100 MHz. The graphics back end IC 212 is coupled to the graphics rendering and memory controller IC 218 by a fourth bus 230. In the preferred embodiment of the present invention, BUS 230 is 64 bits wide and carries signals cycled at 133 MHz. The input/output IC 210 is coupled to the graphics rendering and memory controller IC 218 by a fifth bus 232. In the preferred embodiment of the present invention, BUS 232 is 32 bits wide and carries signals cycled at 133 MHz.

Detailed Description Text (9):
With reference to FIGS. 2A and 2B, GBE interface 232 buffers and transfers display data from unified system memory 202 to the graphics back end IC 212 in 16.times.32-byte

bursts. GBE interface 232 buffers and transfers video capture data from the graphics back end IC 212 to unified system memory 202 in 16.times.32-byte bursts. GBE interface 232 issues GBE interrupts to CPU/IPCE interface 234. BUS 228, shown in both FIG. 2A and FIG. 2B, couples GBE interface 232 to the graphics back end IC 212 (FIG. 2A). The input/output interface 236 buffers and transfers data from unified system memory 202 to the input/output IC 210 in 8.times.32-byte bursts. The input/output interface 236 buffers and transfers data from the input/output IC 210 to unified system memory 202 in 8.times.32-byte bursts. The input/output interface 236 issues the input/output IC interrupts to CPU/IPCE interface 234. BUS 230, shown in both FIG. 2A and FIG. 2B, couples the input/output interface 236 to the input/output IC 210 (FIG. 2A). A bus, BUS 224, provides coupling between CPU/IPCE interface 234 and CPU 206 and the image processing and compression and expansion IC 216.

Detailed Description Text (10):
With reference to FIG. 2A, the memory controller 214 is the interface between memory system clients (CPU 206, rendering engine 208, input/output IC 210, graphics back end IC 212, image processor 214, and data compression/expansion device 215) and the unified system memory 202. As previously mentioned, the memory controller 214 is coupled to unified system memory 202 via HBWMD BUS 225 which allows fast transfer of large amounts of data to and from unified system memory 202. Memory clients make read and write requests to unified system memory 202 through the memory controller 214. The memory controller 214 converts requests into the appropriate control s equences and passes data between memory clients and unified system memory 202. In the preferred embodiment of the present invention, the memory controller 214 contains two pipeline structures, one for commands and another for data. The request pipe has three stages, arbitration, decode and issue/state machine. The data pipe has only one stage, ECC. Requests and data flow through the pipes in the following manner. Clients place their requests in a queue. The arbitration logic looks at all of the requests at the top of the client queues and decides which request to start through the pipe. From the arbitration stage, the request flows to the decode stage. During the decode stage, information about the request is collected and passed onto an issue/state machine stage.

Detailed Description Text (11):
With reference to FIG. 2A, the rendering engine 208 is a 2-D and 3-D graphics coprocessor which can accelerate rasterization. In a preferred embodiment of the present invention, the rendering engine 208 is also cycled at 66 MHz and operates synchronously to the unified system memory 202. The rendering engine 208 receives rendering parameters from the CPU 206 and renders directly to frame buffers stored in the unified system memory 202 (FIG. 2A). The rendering engine 208 issues memory access requests to the memory controller 214. Since the rendering engine 208 shares the unified system memory 202 with other memory clients, the performance of the rendering engine 208 will vary as a function of the load on the unified system memory 202. The rendering engine 208 is logically partitioned into four major functional units: a host interface, a pixel pipeline, a memory transfer engine, and a memory request unit. The host interface controls reading and writing from the host to programming interface registers. The pixel pipeline implements a rasterization and rendering pipeline to a frame buffer. The memory transfer engine performs memory bandwidth byte aligned clears and copies on both linear buffers and frame buffers. The memory request unit arbitrates between requests from the pixel pipeline and queues up memory requests to be issued to the memory controller 214.

Detailed Description Text (12):
The computer system 200 includes dynamic memory allocation of virtual pixel buffers in the unified system memory 202. Pixel buffers include frame buffers, texture maps, video maps, image buffers, etc. Each pixel buffer can include multiple color buffers, a depth buffer, and a stencil buffer. In the present invention, pixel buffers are allocated in units of contiguous memory called tiles and address translation buffers are provided for dynamic allocation of pixel buffers.

Detailed Description Text (14):
With reference to FIG. 3B, an illustration is shown of an exemplary pixel buffer 302 according to the present invention. In the computer system 200 of the present invention, translation hardware maps virtual addresses of pixel buffers 302 to physical memory locations in unified system memory 202. Each of the computational units of the computer system 200 (image processing and compression and expansion IC, 212, graphics back end IC 212, The input/output IC 210, and rendering engine 208) includes translation hardware for mapping virtual addresses of pixel buffers 302 to physical memory locations in unified system memory 202. Each pixel buffer 302 is partitioned into n.sup.2 tiles 300, where n is an integer. In a preferred embodiment of the present

invention, n=4.

Detailed Description Text (15):
The rendering engine 208 supports a frame buffer address translation buffer (TLB) to translate frame buffer (x,y) addresses into physical memory addresses. This TLB is loaded by CPU 206 with the base physical memory addresses of the tiles which compose a color buffer and the stencil-depth buffer of a frame buffer. In a preferred embodiment of the present invention, the frame buffer TLB has enough entries to hold the tile base physical memory addresses of a 2048.times.2048 pixel color buffer and a 2048.times.2048 pixel stencil-depth buffer. Therefore, the TLB has 256 entries for color buffer tiles and 256 entries for stencil-depth buffer tiles.

Detailed Description Text (16):
Tiles provide a convenient unit for memory allocation. By allowing tiles to be scattered throughout memory, tiling makes the amount of memory which must be contiguously allocated manageable. Additionally, tiling provides a means of reducing the amount of system memory consumed by frame buffers. Rendering to tiles which do not contain any pixels pertinent for display, invisible tiles, can be easily clipped out and hence no memory needs to be allocated for these tiles. For example, a 1024.times.1024 virtual frame buffer consisting of front and back RGBA buffers and a depth buffer would consume 12 Mb of memory if fully resident. However, if each 1024.times.1024 buffer were partitioned into 64 (128.times.128) tiles of which only four tiles contained non-occluded pixels, only memory for those visible tiles would need to be allocated. In this case, only 3 MB would be consumed.

Detailed Description Text (17):
In the present invention, memory system clients (e.g., CPU 206, rendering engine 208, input/output IC 210, graphics back end IC 212, image processor 214, and data compression/expansion device 215) share the unified system memory 202. Since each memory system client has access to memory shared by each of the other memory system clients, there is no need for transferring data from one dedicated memory unit to another. For example, data can be received by the input/output IC 210, decompressed (or expanded) by the data compression/expansion device 215, and stored in the unified system memory 202. This data can then be accessed by the CPU 206, the rendering engine 208, the input/output IC 210, the graphics back end IC 212, or the image processor 214. As a second example, the CPU 206, the rendering engine 208, the input/output IC 210, the graphics back end IC 212, or the image processor 214 can use data generated by the CPU 206, the rendering engine 208, the input/output IC 210, the graphics back end IC 212, or the image processor 214. Each of the computational units (CPU 206, input/output IC 210, the graphics back end IC 212, the image processing and compression and expansion IC 216, the graphics rendering and memory controller IC 218, and the data compression/expansion device 215) has translation hardware for determining the physical addresses of pixel buffers as is discussed below.

Detailed Description Text (18):
There are numerous video applications for which the present invention computer system 200 provides functional advantages over prior art computer system architectures. These applications range from video conferencing to video editing. There is significant variation in the processing required for the various applications, but a few processing steps are common to all applications: capture, filtering, scaling, compression, blending, and display. In operation of computer system 200, input/output IC 210 can bring in a compressed stream of video data which can be stored into unified system memory 202. The input/output IC 210 can access the compressed data stored in unified system memory 220, via a path through the graphics rendering and memory controller IC 218. The input/output IC 210 can then decompress the accessed data and store the decompressed data into unified system memory 202. The stored image data can then be used, for example, as a texture map by rendering engine 208 for mapping the stored image onto another image. The resultant image can then be stored into a pixel buffer which has been allocated dynamically in unified system memory 202. If the resultant image is stored into a frame buffer, allocated dynamically in unified system memory 202, then the resultant image can be displayed by the graphics back end IC 212 or the image can be captured by writing the image back to another pixel buffer which has been allocated dynamically in unified system memory 202. Since there is no necessity of transferring data from one dedicated memory unit to another in computer system 200, functionality is increased.

Detailed Description Text (19):
In the preferred embodiment of the present invention, unified system memory 202 of FIG. 2A is implemented using synchronous DRAM (SDRAM) accessed via a 256-bit wide memory

data bus cycled at 66 MHz. A SDRAM is made up of rows and columns of memory cells. A row of memory cells is referred to as a page. A memory cell is accessed with a row address and column address. When a row is accessed, the entire row is placed into latches, so that subsequent accesses to that row only require the column address. Accesses to the same row are referred to as page accesses. In a preferred embodiment of the present invention, unified system memory 202 provides a peak data bandwidth of 2.133 Gb/s. Also, in a preferred embodiment of the present invention, unified system memory 202 is made up of 8 slots. Each slot can hold one SDRAM DIMM. A SDRAM DIMM is constructed from 1M.times.16 or 4M.times.16 SDRAM components and populated on the front only or the front and back side of the DIMM. Two DIMMs are required to make an external SDRAM bank. 1M.times.16 SDRAM components construct a 32 Mbyte external bank, while 4M.times.16 SDRAM components construct a 128 Mbyte external bank. unified system memory 202 can range in size from 32 Mbytes to 1 Gbyte.

Detailed Description Text (26):
The data and mask are latched in the data pipe and flow out to the unified system memory 202 on memmask.sub.-- out and memdata2mem.sub.-- out. From the data and mask, the ECC and ECC mask are generated and sent to the unified system memory 202 across eccmask and ecc.sub.-- out. The memdataoe signal is used to turn on the memory bus drivers. Data and ECC from the unified system memory 202 come in on the memdata2client.sub.-- in and ecc.sub.-- in busses. The ECC is used to determine if the incoming data is correct. If there is a one bit error in the data, the error is corrected, and the corrected data is sent to the memory client. If there is more than one bit in error, the CPU 206 is interrupted, and incorrect data is returned to the memory client. Ras.sub.-- n, cas.sub.-- n, we.sub.-- n and cs.sub.-- n are control signals for the unified system memory 202.

Detailed Description Text (27):
With reference to FIG. 8, a timing diagram is shown for an exemplary write to a new page performed by the unified system memory 202. With reference to FIG. 9, a timing diagram is shown for an exemplary read to a new page performed by the unified system memory 202. A read or write operation to the same SDRAM page is the same as the operation shown in FIGS. 8 and 9, except a same page operation does not need the precharge and activate cycles.

Detailed Description Text (28):
A request pipe is the control center for the memory controller 204. Memory client requests are placed in one end of the pipe and come out the other side as memory commands. The memory client queues are at the front of the pipe, followed by the arbitration, then the decode, and finally the issue/state machine. If there is room in their queue, a memory client can place a request in it. The arbitration logic looks at all of the requests at the top of the memory client queues and decides which request to start through the request pipe. From the arbitration stage, the request flows to the decode stage. During the decode stage, information about the request is collected and passed onto the issue/state machine stage. Based on this information, a state machine determines the proper sequence of commands for the unified system memory 202. The later portion of the issue stage decodes the state of the state machine into control signals that are latched and then sent across to the unified system memory 202. A request can sit in the issue stage for more than one cycle. While a request sits in the issue/state machine stage, the rest of the request pipe is stalled. Each stage of the request pipe is discussed herein.

Detailed Description Text (40):
The unified system memory 202 is made up of 8 slots. Each slot can hold one SDRAM DIMM. A SDRAM DIMM is constructed from 1M.times.16 or 4M.times.16 SDRAM components and populated on the front only or the front and back side of the DIMM. Two DIMMs are required to make an external SDRAM bank. 1M.times.16 SDRAM components construct a 32 Mbyte external bank, while 4M.times.16 SDRAM components construct a 128 Mbyte external bank. The memory system can range in size from 32 Mbytes to 1 Gbyte.

Detailed Description Text (57):
The main functions of the data pipe are to: (1) move data between a memory client and the unified system memory 202, (2) perform ECC operations and (3) merge new byte from a memory client with old data from memory during a read-modify-write operation. Each of these functions is described below.

Detailed Description Text (60):
pipe. The data is held in the ECC stage and flows out to the unified system memory 202 until the request is retired in the request pipe.

Detailed Description Text (61):
Incoming read data is latched in the data pipe, flows through the ECC correction logic and then is latched again before going on the Memdata2client.sub.-- out bus. The request pipe knows how many cycles the unified system memory 202 takes to return read responses data. When the read response data is on the Memdata2client.sub.-- out bus, the request pipe asserts clientres.rdrdy.

Detailed Description Paragraph Table (2):
clientreq.adr internal 25 in address of request only clientreq.msg internal 7 in message sent with request only clientreq.v internal 1 in 1 - valid only 0 - not valid clientreq.ecc internal 1 in 1 - ecc is valid only 0 - ecc not valid clientres.gnt internal 1 out 1 - room in client queue only 0 - no room clientres.wrrdy internal 1 out 1 - MC is ready for write data only 0 - MC not ready for write data clientres.rdrdy internal 1 out 1 - valid read data only 0 - not valid read data clientres.oe internal 1 out 1 - enable client driver only 0 - disable client driver clientres.rdmsg internal 7 out read message sent with read only data clientres. wrmsg internal 7 out write message sent with wrrdy only memdata2- internal 256 out memory data from client mem.sub.-- in only going to unified system memory memmask.sub.-- in internal 32 in memory mask from client only going to unified system memory 0 - write byte 1 - don't write byte memmask.sub.-- in (0) is matched with memdata2mem.sub.-- in (7:0) and so on. memdata2- internal 256 out memory data from unified client.sub.-- out only system memory going to the client

CLAIMS:

1. A computer system comprising:

a memory controller;

a graphics rendering engine coupled to said memory controller;

a CPU coupled to said memory controller;

an image processor coupled to said memory controller;

a data compression/expansion device coupled to said memory controller;

an input/output device coupled to said memory controller;

a graphics back end device coupled to said memory controller;

a system memory, coupled to said memory controller via a high bandwidth data bus, said system memory providing read/write access, through said memory controller, for memory clients including said CPU, said input/output device, said graphics back end device, said image processor, said data compression/expansion device, said rendering engine, and said memory controller, wherein said memory controller is the interface between said memory clients and said system memory; and

translation hardware for mapping virtual addresses of pixel buffers to physical memory locations in said system memory wherein said pixel buffers are dynamically allocated as tiles of physically contiguous memory.

11. A computer system comprising:

a graphics rendering engine and a memory controller implemented on a first IC;

a CPU coupled to said first IC;

an image processor coupled to said first IC;

a data compression/expansion device coupled to said first IC;

an input/output device coupled to said first IC;

a graphics back end device coupled to said first IC;

a system memory, coupled to said first IC via a high bandwidth data bus, said system memory providing read/write access, through said first IC, for memory clients including

said CPU, said input/output device, said graphics back end device, said image processor , said data compression/expansion device, said rendering engine, and said memory controller, wherein said memory controller is the interface between said memory clients and said system memory; and

translation hardware for mapping virtual addresses of pixel buffers to physical memory locations in said system memory wherein said pixel buffers are dynamically allocated as tiles of physically contiguous memory.

21. A computer system comprising:

a CPU;

an input/output device;

a graphics back end unit;

a first IC including an image processor and a data compression and expansion device integrated therein;

a second IC including a graphics rendering engine and a memory controller device integrated therein;

a system memory which allows read/write access for memory clients including said CPU, said input/output device, said graphics back end device, said image processor, said data compression/expansion device, said rendering engine, and said memory controller, wherein said memory controller is the interface between said memory clients and said system memory;

a high bandwidth data bus for transferring data between said system memory and said second IC; and

translation hardware for mapping virtual addresses of pixel buffers to physical memory locations in said system memory wherein said pixel buffers are dynamically allocated as tiles of physically contiguous memory.